Fence Placement for Legacy Data-Race-Free Programs via Synchronization Read Detection

Fence Placement for Legacy Data-Race-Free Programs via Synchronization Read Detection

Andrew J. McPherson
University of Edinburgh
ajmcpherson@ed.ac.uk

Vijay Nagarajan
University of Edinburgh
vijay.nagarajan@ed.ac.uk

Susmit Sarkar
University of St. Andrews
ss265@st-andrews.ac.uk

Marcelo Cintra
Intel
marcelo.cintra@intel.com

Abstract

Shared-memory programmers traditionally assumed Sequential Consistency (SC), but modern systems have relaxed memory consistency. Here, the trend in languages is towards Data-Race-Free (DRF) models, where, assuming annotated synchronizations and the program being well-synchronized by those synchronizations, the hardware and compiler guarantee SC. However, legacy programs lack annotations, so even well-synchronized (legacy DRF) programs aren't recognized. For legacy DRF programs, we can significantly prune the set of memory orderings determined by automated fence placement, by automatically identifying synchronization reads. We prove our rules for identifying them conservative, implement them within LLVM, and observe a 30% average performance improvement over previous techniques.

1. Introduction

1.1 The Problem

A memory consistency model is at the heart of shared memory concurrency, and specifies the value that each read in the program can return. Sequential consistency (SC) [26] in which each read returns the last value written to that location in a global order found by interleaving the actions of each thread, is arguably the most intuitive of memory models [11, 24, 28, 37].

Unfortunately, as is now well-known, modern hardware does not provide SC to the programmer. Instead, different hardware architectures produce different varieties of relaxed consistency behavior [2]. Also, an agnostic compiler could perform optimizations which could violate SC.

The primary means by which the compiler can provide support is to insert appropriate fences to enforce sufficient orderings to restore SC. Each processor architecture provides different fences to enforce various types of orderings. The challenge is to insert sufficient fences to restore SC, while at the same time not inserting too many. Fences are expensive, since they limit many of the optimization opportunities available to hardware because of the relaxed memory consistency. Indeed, placing fences between every pair of accesses would guarantee SC, but would be far too expensive.

The starting point of understanding the required placement of fences is the seminal *Delay-set analysis* of Shasha and Snir [36]. They observed that to ensure SC, it is not necessary to order all pairs of accesses. Only conflicting pairs of accesses (the delay sets) that can potentially lead to SC violations need to be ordered – where conflicting accesses are two accesses to the same address, at least one of which is a write. The memory orderings produced by Delay-set analysis are then subject to *fence minimization* [28], which seeks to minimize the number of fences required to enforce the above memory orderings.

One major issue that limits the practicality of Delay-set analysis is its reliance on alias analysis which is notoriously imprecise for programs that make heavy use of pointers. In addition to this, scalability is also an issue for large programs. To overcome the scalability issue, approximations of Delay-set analysis using escape analysis have been developed, notably by the the Pensieve project [17, 38]. More recently, attempts have also been made to address the scalability issue without resorting to escape analysis [5] – although recursion and dynamic thread creation continues to limit applicability. For either approach however, the imprecision issue remains unresolved, even with state-of-the-art alias analysis. This causes Delay-set analysis to produce a large number of superfluous orderings for real-world programs [2, 29, 37].

1.2 Our Approach

We take a fresh look at fence placement. Our point of departure is that we do not seek to enforce SC for the general case.

Instead, we insert sufficient fences to ensure that those memory accesses that are race free[1] in the SC world continue to be race free in the relaxed world. To put it succinctly, we guarantee SC behavior only for race free accesses.

Our approach is based on the realization that SC (which strongly orders all accesses) is not an end in itself to programmers; rather it is enough for programmers to have SC semantics only for synchronization accesses (where synchronization accesses are those accesses that are used to guard other data accesses from racing). Therefore, it suffices if we identify such synchronization accesses and provide SC semantics for only those accesses. In order to understand this better, let us consider the two examples shown in Figures 1(a) and 1(b).

In the producer-consumer example shown in Figure 1(a), the programmer synchronizes using the flag variable, to ensure that the read b_2 returns the value produced by a_1 (and not the old value). In this example, accesses a_2 and b_1 are synchronization accesses. Therefore, providing SC semantics to these accesses ensures that b_2 reads the correct value. The second example, shown in Figure 1(b), is a piece of code similar to that found in a relaxation solver [13, 19], in which the four accesses involved are unsynchronized accesses (by design). Here, it is permissible for the accesses in either thread to be reordered, e.g., for the read of x in P2 to return a stale value (occurring before a_1 in P1) while b_1 reads the value written by a_2. In other words, they are data races, albeit benign in this case. Therefore, providing SC semantics to such unsynchronized accesses is not required.

(a)

P1	P2
$a_1 : data = 1;$	
$a_2 : flag = 1;$	b_1 :while$(flag == 0);$
	$b_2 : x = data;$

(b)

P1	P2
$a_1 : x = C_1;$	
$a_2 : y = C_2;$	$b_1 : local_2 = y;$
	$b_2 : local_1 = x;$

Figure 1. Examples of well-synchronized (a), and not well-synchronized (b) programs.

Although we do not promise SC in general, it is important to note that our approach guarantees SC for well-synchronized programs i.e., legacy data-race-free programs[2]. Figure 1(a) is an example of a well-synchronized program, whereas Figure 1(b) is not.

Our approach is similar in spirit to DRF (data-race-free) programming models, which form the basis of recent concurrent programming language models, such as the C11 concurrency model [7, 10] and the Java Memory Model specification [30]. This is a programming model which gives semantics to only DRF programs: programs in which synchronization operations are correctly labelled and the program is well-synchronized using those operations. In return for this discipline the system (hardware + compiler) guarantees SC. However, legacy programs lack the distinction between data and synchronization. Our approach automatically discovers synchronization operations for such legacy programs.

1.3 Our Solution

We look for ways to conservatively identify synchronization operations. If we can be relatively precise, we can prune unnecessary orderings found by more traditional approaches. The existing fence minimization techniques can then be applied on the pruned orderings to achieve improved performance. An alternative application would be to use this identification to provide minimal annotations to make the program DRF, such that a compliant compiler and the hardware will prevent incorrect reorderings.

We have identified two signatures, at least one of which must be fulfilled for a read to be a synchronization, i.e., an acquire operation:

- **Control acquire**: a read feeds its value to a predicate tested for in a branch in its forward slice.
- **Address acquire**: a read provides the address value for a subsequent data access that the read (acquire) protects.

We formally prove that at least one of these must hold for a read to be an acquire. The second signature (address acquire) is less prevalent, and in particular is observed to appear along with the first signature (control acquire) in all cases in our experiments. We do not improve the identification of releases and, as in Pensieve, conservatively consider every shared write (escaping write) to be a release.

To evaluate the significance of our contribution, we next design and implement practical algorithms for identifying the acquires. Our simpler first algorithm (**Fast**) detects only control acquires, and does not do interprocedural flow analysis (which is expensive). This does mean that the algorithm theoretically does not detect all acquiring reads. In particular, it does not detect cases where the acquiring read and the branch (both of which intuitively form the acquire) are split

[1] A memory access is said to be race free if in all legal SC executions, it is ordered with its conflicting accesses in each execution, via the ordering chain introduced in section 3 (following [20])

[2] More formally, these refer to a class of programs whose behavior is characterized by values returned by only those reads that are race free under SC.

across two functions[3]. We believe this will only rarely if ever be violated. In all our experiments we never see such a split, though contrived programs can be written.

Fast will also not detect address acquires. Again, in all our experiments, we have never seen an address acquire which is not also a control acquire. However, for completeness, we also develop a conservative variant of our algorithm (**Safe**). This variant detects address acquires in addition to control acquires.

We implemented our analysis in LLVM and applied it to the SPLASH-2 benchmark suite and a set of lock-free programs. Our experimental results show that on average, Fast reduces the number of orderings considered by 66% on average. Applying a fence minimization technique, this translates to an average of 62% fewer fences on x86-TSO and up to 2.64x speedup over an existing practical technique. Safe on average reduces orderings considered by 32%, fences placed by 27% and produces speedup of up to 1.54x.

The contributions of this paper are:

1. We improve fence insertion for legacy programs by discovering synchronization read operations.
2. We prove that for all the necessary orderings (*essential orderings*) involving a synchronization read, the read has to satisfy at least one of two specific signatures: (a) that there is a conditional branch whose condition depends on the value returned by the read in the forward slice of the read. (b) that a read provides the address for a subsequent access that would otherwise be unknown.
3. We propose two practical algorithms: **Fast** that detects only control acquires and **Safe** that detects both address and control acquires. Both algorithms work in the presence of pointers.
4. We implement our algorithm within LLVM, and observe an average of 62% fewer fences and up to 2.64x speedup over an existing practical technique with the simpler algorithm, and an average of 27% fewer fences and up to 1.54x speedup with the conservative algorithm.

2. Our Approach

2.1 Fence Placement: Background

The starting point of understanding the required placement of fences is Shasha and Snir's Delay-set analysis. Its key insight is that not all pairs of memory accesses need to be ordered to ensure SC. Only pairs of memory accesses that conflict with accesses from other threads, potentially leading to (minimal) SC violations known as *critical cycles* need to be ordered. Identifying such critical cycles however, presents a scalability issue on real-world programs (with pointers, recursion etc.), as it relies on heavyweight interprocedural static analysis. To overcome this, practical tools such as Pensieve [17, 38], approximate Delay-set analysis.

This conservative approximation is attained by such tools in a two step process. Firstly, a conservative thread-escape analysis is performed on each access in a function, to determine a set of potentially escaping accesses, E. Secondly, for $u, v \in E$, if analysis of the control flow graph shows that v can occur after u, then an ordering, $u \rightarrow v$, is recorded.

While this does generate a correct set of orderings, it produces a large number of false positives due to the thread-escape analysis being necessarily conservative. In practice this means that all references to memory that cannot be proven to be restricted to the local function, must be marked as potentially escaping.

Once a set of orderings has been identified, these orderings are fed as input to a fence minimization algorithm. Such an algorithm will determine where to minimally place fences to ensure that all the orderings are enforced. It may also distinguish between types of orderings, to minimize the cost of enforcement. This can be achieved by using different types of fences or compiler directives, depending on the memory consistency model of the target architecture. For example, x86-TSO only requires orderings of the type $w \rightarrow r$ to be enforced with full memory fences, as other orderings are enforced by the hardware. These other orderings however, still have to be preserved during the compilation (optimization) process.

2.2 Fence Placement for DRF Programs

Now let us consider fence placement for a DRF program. Recall that in a DRF program, synchronization is achieved using special memory operations – a write known as a *release* and a read known as an *acquire* – such that there are no races amongst data operations. This implies that given such a well-synchronized program without data races, enforcing the orderings defined in Table 1 is sufficient to ensure correctness [1].

In more detail, the first rule requires that all accesses to shared data must be performed before a release. Similarly, the second rule requires that all accesses to shared data must be performed only after an acquire. These two, combined with the third rule, ordering all acquires and releases, ensures correctness.

With precise information as to which of the reads (writes) are acquires (releases), determining the minimal set of required orderings is trivial. Specifically, orderings that do not conform to one of the definitions in Table 1, could be safely ignored. The set of required orderings could then be fed as input to a fence minimization algorithm.

[3] Note that the data accesses which the acquire protects are subject to no such assumption, and can be located in a separate function.

[4] Weaker models which relax some of these requirements, such as RC_{PC} [2] in hardware and C11 [7, 10] at the language level also exist.

$r/w \rightarrow w_{rel}$	All reads and writes before the release (in program order) should be ordered before the release
$r_{acq} \rightarrow r/w$	All reads and writes after the acquire (in program order) should be ordered after the acquire
$w_{rel}/r_{acq} \rightarrow w_{rel}/r_{acq}$	All synchronization operations should be ordered among themselves[4]

Table 1. Sufficient orderings for correctness in a DRF program

Legacy DRF Code		Delay-set Fence Placement		Pruned Orderings Fence Placement	
P1	P2	P1	P2	P1	P2
$a_1 : x =$	$b_1 : *p1 =$	$a_1 : x =$	$b_1 : *p1 =$	$a_1 : x =$	$b_1 : *p1 =$
		—$(F1)$—	—$(F3)$—		
$a_2 := y$	$b_2 := *p2$	$a_2 := y$	$b_2 := *p2$	$a_2 := y$	$b_2 := *p2$
		—$(F2)$—		—$(F2)$—	
$a_3 : flag = 1$	b_3 : while$(flag! = 1)$;	$a_3 : flag = 1$	b_3 : while$(flag! = 1)$;	$a_3 : flag = 1$	b_3 : while$(flag! = 1)$;
			—$(F4)$—		—$(F4)$—
	$b_4 : y =$		$b_4 : y =$		$b_4 : y =$
			—$(F5)$—		
	$b_5 := x$		$b_5 := x$		$b_5 := x$

Figure 2. An Example of (full) fence placement on legacy DRF code for Delay-set and pruned orderings.

2.3 Identifying Acquires for Legacy DRF

There exists however, a large body of (legacy) code which is correctly synchronized, but the distinction between a read (r) and an acquiring read (r_{acq}), and a write (w) and a release (w_{rel}) is not made explicit by the programmer. We call such programs Legacy DRF.

One way to perform fence placement for such programs is to treat it like a general multithreaded program, i.e., use Delay-set analysis (or its conservative approximation) followed by fence minimization techniques. Our key insight is that we can do better if we can conservatively identify synchronization operations. In this paper, we focus on detecting acquires.

We prove that for a read to be an acquire it must match at least one of two signatures. The first is that there exists a branch whose predicate is data dependent on the read, in the forward slice of that read. The second is that the read provides the address value for a subsequent data access that the read protects. Any read that fails to satisfy at least one of these signatures cannot be an acquire.

Intuitively, an acquire is a read which determines if shared data can be accessed. This necessarily involves either checking the value read and acting upon it (the first signature), or providing the address of data, which would otherwise be inaccessible (the second signature). A formal proof of these assertions can be found in Section 3.

By applying the two signatures to every read which may be thread-escaping, we determine a subset that includes every potential acquire.

Having identified a conservative subset of the shared reads as potential acquires, we are able prune the orderings. Starting from the set of orderings given by Delay-set analysis (or its approximation that uses escape analysis), we prune all those orderings which do not adhere to one of the definitions in Table 1. Despite not identifying a subset of the shared writes and therefore having to consider all shared writes as releases, we are still able to prune a number of potentially expensive orderings.

Specifically, any ordering of the form $r_1 \rightarrow r_2$ requires at least r_1 to be an acquire to avoid being pruned, i.e., it must be of the form $r_{acq} \rightarrow r$. Similarly, any ordering of the form $w_1 \rightarrow r_2$ requires r_2 to be an acquire to avoid being pruned, i.e., of the form $w \rightarrow r_{acq}$.

This reduced number of orderings is provided as (an improved) input to a fence minimization algorithm, resulting in a much reduced number of fences.

2.4 An Example

To illustrate the impact of pruning orderings, we now demonstrate the application of Delay-set analysis to a section of legacy DRF code and the fences that this would require. Then, using the acquire signatures and applying the pruning rules defined above, we determine the reduced set of fences required to enforce the remaining orderings.

In Figure 2, we present a section of legacy DRF code which contains a busy-waiting synchronization. For the purposes of this example we assume that alias analysis has determined that $*p1$ and $*p2$ may potentially alias with both x and y, but not $flag$. If one were to apply Delay-set analysis, the following orderings would be determined to avoid the following critical cycles:

- $a_1 \rightarrow a_3, b_3 \rightarrow b_5$: to avoid $(a_1, a_3, b_3, b_5, a_1)$.
- $a_2 \rightarrow a_3, b_3 \rightarrow b_4$: to avoid $(a_2, a_3, b_3, b_4, a_2)$.
- $a_1 \rightarrow a_2, b_4 \rightarrow b_5$: to avoid $(a_1, a_2, b_4, b_5, a_1)$.
- $a_1 \rightarrow a_2, b_1 \rightarrow b_2$: to avoid $(a_1, a_2, b_1, b_2, a_1)$.

In the final cycle our assumption regarding $*p1$ and $*p2$ potentially aliasing with x and y but not $flag$ comes into play.

Using these orderings as input to a fence minimization algorithm, 5 (full) fences are required to be placed to enforce the orderings. Placement of these fences is shown as "Delay-set Fence Placement" in Figure 2.

Pruning the orderings by applying the signatures defined in Section 2.3, we find that only the following remain:

- $a_1 \rightarrow a_3, b_3 \rightarrow b_5$: to avoid $(a_1, a_3, b_3, b_5, a_1)$.
- $a_2 \rightarrow a_3, b_3 \rightarrow b_4$: to avoid $(a_2, a_3, b_3, b_4, a_2)$.

Of the orderings which have been pruned: $a_1 \rightarrow a_2$, $b_1 \rightarrow b_2$ and $b_4 \rightarrow b_5$ are not required as none of a_2, b_2 or b_5 are acquires. Using this reduced set of orderings as input to the same fence minimization algorithm, only 2 (full) fences are required to be placed. These fences are shown as "Pruned Orderings Fence Placement" in Figure 2.

$F1, F3$ and $F5$ are no longer required and have been removed. However, $F2$ and $F4$ are still required. Together they prevent $(a_1, a_3, b_3, b_5, a_1)$ and $(a_2, a_3, b_3, b_4, a_2)$, with $F2$ enforcing $a_1 \rightarrow a_3$ and $a_2 \rightarrow a_3$, and $F4$ enforcing $b_3 \rightarrow b_4$ and $b_3 \rightarrow b_5$.

In summary, we expect our signatures to considerably reduce the number of orderings that need to be enforced. With reference to our example, there are three major benefits.

- Acquire detection allows us to avoid enforcing many orderings that are not necessary (e.g., data $\rightarrow$ data orderings such as $a_1 \rightarrow a_2$ and $b_4 \rightarrow b_5$), since the program is well-synchronized.
- The inherent imprecision of Delay-set analysis (or its approximation) in the presence of pointers results in the enforcement of orderings which are not necessary. Acquire detection allows us to prune some of these orderings (e.g., $b_1 \rightarrow b_2$).
- This reduction in the number of orderings, allows a fence minimization algorithm to place fewer fences, (in this case, not placing $F1, F3$ and $F5$).

3. Correctness of Acquire Signatures

In this section we formally prove the basis of our assertions above, that is, a synchronization read (acquire) matches (at least) one of two signatures. One is that in its forward slice, there must be a conditional dependent on the value returned by the read. The other is that the acquire reads a value determining the address of a subsequent access.

Language For concreteness, we define our programming language to be a simple multi-threaded "while" language with pointers. Expressions e are pure, defined as making no shared-memory loads or stores, though local variables (marked with an r are allowed. Statements then can dereference pointers, load from and store to shared-memory locations, either explicitly or via pointers. The language is presented in Figure 3.

This tiny language captures all the essential features needed for our results. Note that in comparison to a full-scale language such as C, key simplifications are that all shared-memory loads and stores from a single thread are explicitly sequenced, and that function calls and returns are ignored. We also ignore read-modify-writes, but these can easily be added to the proof below, by considering them to be a read followed by a write to the same location.

Shared locations	x;		Local variables r
Expressions	e	$::=$	$\&x \mid r \mid e + e \mid \ldots$
Statements	s	$::=$	$x := e \mid r := x$
			$\mid r := *e \mid *e := e$
			$\mid$ skip $\mid$ if (e) then s else s
			$\mid$ while (e) do s
			$\mid s; s \mid s \| s \mid \ldots$

Figure 3. The programming language for proofs

Intended Behavior Given a program in the above language, we assume that there is some intended marking of accesses (shared-memory loads and stores) into data and synchronization accesses. Data accesses are programmer-intended accesses; more formally, the behavior intended by the programmer is defined by the values read by the data reads. The rest of the accesses are assumed to be synchronization accesses; these are assumed to be written only to make sure there are no races on the data accesses. Following standard practice, we call synchronization reads *acquire* reads and synchronization writes *release* writes.

Behavior under SC A *sequentially consistent execution* is an execution trace (a linear order of read and write actions) which is a free interleaving of thread-wise actions, such that actions belonging to any thread appear in the execution trace in the order they occur in that thread, and each memory read reads the value of the last write to that location in the trace. Note that in general, a single access in the program might lead to one or more actions in the trace (due to loops), or none (in case of a conditional). There is a straightforward way of associating each action in the trace to at most one program access, and we associate the corresponding kind (data or synchronization) of program access to the actions. Of course, because there might be several possible interleavings, a program has a set of allowed sequentially consistent executions. For each such execution, we intuitively consider the results of the execution to be the values returned by the data reads. We formally consider the intended behavior of the program to be the set of data read actions of any possible sequentially consistent execution.

Behavior under relaxed consistency A program actually executes not on a sequentially consistent machine but on a machine with relaxed consistency. We follow the approach of Adve and Hill [3] (the approach of Gharachorloo [22] is very similar), and define that a program is correct iff it has no more behavior in a relaxed consistency setting than in the sequentially consistent world.

We define happens-before following Gharachorloo [20] by first defining conflict order and program order. Define *conflict order* $\xrightarrow{con}$ to be an order relation between conflicting actions in an execution (the order says one happens before the other), where two actions conflict if they are to the

same address and at least one is a write. In particular, a write is conflict-ordered before a read if the read reads from that write. Also, there is an obvious *program order* relation $\xrightarrow{po}$ between actions from the same thread.

Given two actions u and v, u happens-before v (written $u \xrightarrow{hb} v$) in that execution if either $u \xrightarrow{po} v$ or $u \xrightarrow{po} w_1 \xrightarrow{con} r_1 \xrightarrow{po} w_2 \xrightarrow{con} r_2 \ldots w_n \xrightarrow{con} r_n \xrightarrow{po} v$. We consider only executions in which each synchronization read reads from the last write to that location in happens-before. The behavior of a program is determined by the data reads (value and location) of all such executions.

Well synchronized programs We call a program (legacy) data-race-free if in all executions (where synchronization reads read from the last write in happens-before as above), all conflicting data actions are ordered by $\xrightarrow{hb}$. It has been proved [3, 22] that data-race-free programs have no more behavior in this sense than sequentially consistent behavior of the same program. However, since legacy programs do not have explicit markings of data and synchronization, and to avoid confusion with the standard data-race-free notion, we equivalently call legacy data-race-free programs well-synchronized.

Ordering edges: Essential and Non-essential We call a program order edge *essential* if ignoring that edge allows a data read to read a value not possible under SC, and all other program order edges *non-essential*. Thus enforcing all essential program order edges is sufficient to preserve SC behavior for the data reads.

We now prove a happens-before characterization of essential edges. Specifically, we prove that an edge in a well-synchronized program, i.e. (legacy) data-race-free program, is essential iff ignoring that edge in happens-before defined as above allows an execution with a data race.

LEMMA 1. *For a program which is data-race-free for a certain mapping, and $U \rightarrow V$ a program order edge, the edge is essential iff deleting $U \rightarrow V$ from happens-before allows an execution with a data race involving a read and write.*

Proof Both directions follow easily from unfolding the definitions.

For one direction, ignoring an essential edge allows a data read to read a value not possible under SC. That data read and the write it reads from must be in a data race, since if they are ordered via happens-before, then the read is still possible under SC.

In the other direction, suppose deleting $U \rightarrow V$ from happens-before allows an execution with a data race between a read and a write. Consider that read. Since the program is well-synchronized (that is, no data races before removing that edge), the read could not have read from that write. ■

Intuitively, if we disregard an essential ordering edge, the program is no longer data-race-free, and thus the DRF guarantees of [3] and [22] do not apply. In that case (disregarding essential orderings), there will be data reads observable that are not possible in sequentially consistent executions. This happens-before characterization is easier to prove with, as we can now analyze the shapes of happens-before.

Informal explanation We are now in a position to give the formal proof of our main result, Theorem 2. Before that, to orient the reader, we give the main idea of the proof informally.

The key insight is that if there is an essential ordering involving an acquire, then the acquire must have been guarding a data access; only then will relaxing the above ordering result in a data race (and thus, by Lemma 1, non-SC behavior for the data reads). We illustrate 3 different ways in which an acquire can guard data. The formal proof will essentially say that these are the only cases to consider, which allows us to safely deduce the acquire signatures.

The first way in which an acquire can guard data is illustrated via the classic Producer-Consumer or MP (Figure 4). Here the data access (of x) is guarded by control-dependency, that is, control only flows to it if the (acquire) read of $flag$ reads 1.

MP	
P1	P2
$a_1 : x :=$	
$a_3 : flag := 1$	$b_3 :$ while $(r_1! = 1)\{r_1 := flag\}$
	$b_5 : r = x$

Figure 4. The MP example

The second way is when the value read by the acquire is used to calculate the address touched by the data access (that is, it only reads from the location if the acquire read a certain value). This could happen in the example in Figure 5, an example adapted from Gharachorloo. Here y (analogous to $flag$ above) stores the address of z initially, and the second read on the second thread reads from x only if the prior read reads x (otherwise it reads from z).

MP with Pointers	
Initially $z = 0, y = \&z, x = 0$	
P1	P2
$a_1 : x =$	
$a_3 : y = \&x$	$b_3 : r = y;$
	$b_5 : r_1 = *r$

Figure 5. The MP example with pointer arithmetic

The third possible way is to have some form of mutual exclusion, in which the data access is in a critical region. In this case (seen in the Dekker's example in Figure 6), the data access is prevented from performing in an execution where the synchronization read reads the wrong value.

Formal proofs Given a program, and if we knew the marking into data and synchronization, we call two accesses *potentially racing* if they are on different threads, at least one

Dekker	
P1	P2
$a_1 : x := 1$	$b_1 : y := 1$
$a_2 : \mathsf{if}(y == 0)\{$	$b_2 : \mathsf{if}(x == 0)\{$
$a_3 :$ touch $z\}$	$b_3:$ touch $z\}$

Figure 6. The Dekker Example

of them is a data write, and they are either statically to the same location, or at least one of them is is to a statically unknown location (this can happen if it is to a location derived from a value read before on the same thread).

LEMMA 2. *For two potentially racing accesses U and V in the program, and any legal execution X according to the relaxed consistency model, at least one of the following must happen:*

1. *U and V correspond to two actions which form a data race in X;*
2. *U and V correspond to actions u and v respectively in X that are ordered $u \xrightarrow{po} w_1 \xrightarrow{con} r_1 \xrightarrow{po} w_2 \xrightarrow{con} r_2 \ldots w_n \xrightarrow{con} r_n \xrightarrow{po} v$ in X;*
3. *U and V correspond to actions u and v respectively in X that are to different locations (this can only happen for statically unknown locations);*
4. *at least one of U and V do not correspond to any actions in X;*

Proof Immediate from the definitions of data races and happens-before. ■

Lemma 2 intuitively says that for static program accesses that potentially race, in any execution either there is an actual race, or there is a proper happens-before ordering such as in Figure 4 between the actions corresponding to the race, or one or the other access is to a different locations (such as in Figure 5) or absent altogether (such as in Figure 6).

LEMMA 3. *For all essential orderings which are of the following form:*

1. *R $\rightarrow$ A, where R is an acquire and A is a subsequent access; or*
2. *W $\rightarrow$ R, where W is a write and R is a subsequent acquire,*

the value read from the acquire must feed into:

- *Either a conditional which guards a subsequent access;*
- *Or an address computation which determines the location of a subsequent access.*

Proof Given the essential ordering edge in the premise of the theorem. It can be of two types: R $\rightarrow$ A, or W $\rightarrow$ R. Consider disregarding this ordering edge in happens-before. Since the ordering edge is essential, by Lemma 1 there is a data race in some execution. Call that execution X, and consider the two data accesses U and V involved in the race. Since they correspond to racing actions in an execution, they must be *potentially racing* accesses. Consider the execution Y with the ordering edge present, and otherwise is the same as X, except that because reads may read different values, some actions may not occur or occur with different values in Y than in X. Apply Lemma 2 to the legal execution Y. Then one of the four cases must apply.

Case 1: In Y, U and V correspond to two actions u and v which form a data race. Since the program is assumed data-race-free, and Y is a legal execution, this case cannot occur.

Case 2: In Y, U and V correspond to actions u and v respectively in X that are ordered $u \xrightarrow{po} w_1 \xrightarrow{con} r_1 \xrightarrow{po} w_2 \xrightarrow{con} r_2 \ldots w_n \xrightarrow{con} r_n \xrightarrow{po} v$ in X. The ordering edge in question must occur in this chain. Since there is no W $\rightarrow$ R ordering edge in this chain, the essential ordering edge we are dealing with must be of the form R $\rightarrow$ A. We now see where the action corresponding to R occurs in this chain. It cannot be the first step ($u \xrightarrow{po} w_1$), since u is a data access. It can be r_n in the last step ($r_n \xrightarrow{po} v$), or r_i in an intermediate thread ($r_i \xrightarrow{po} w_{i+1}$). In each case, R reads the value of a synchronization write in this execution Y. Furthermore, v or w_{i+1} respectively is the access A in question. Consider now a different execution where R does not read the value of the same synchronization write. Then it must be the case that either A does not occur, or A exists but accesses a different location, since otherwise the ordering chain does not exist and the program has a race. Thus either R feeds into a conditional guarding A or is used to calculate the address touched by A, as required.

Case 3: U and V correspond to actions u and v respectively in Y that are to different locations.

Since U and V correspond to racing actions u' and v' in X, at least one of the pairs (u, u') and (v, v') must be to different locations. Without loss of generality, let u and u' be to different locations. Then U must be to a statically unknown location, that is in fact different in X and Y. Since X differs from Y in that the essential ordering edge (either R $\rightarrow$ A or W $\rightarrow$ R) is not required, in either case the calculation of the location for U must be derived from the value returned by R.

Case 4: At least one of U and V do not correspond to any actions in Y.

Without loss of generality, let there be no actions corresponding to U in Y. Since U corresponds to an action u in X, U must be guarded by a conditional that is true in X but not in Y. Since X differs from Y in that the essential ordering edge (either R $\rightarrow$ A or W $\rightarrow$ R) is not required, in either case this conditional must be derived from the value returned by R.

■

THEOREM 2. *For all essential orderings involving an acquire R, the value read from the acquire must feed into:*

- *Either a conditional which guards a subsequent access;*

- *Or an address computation which determines the location of a subsequent access*

Proof The possible orderings involving an acquire R are:

Case 1: $R_1 \rightarrow R$, where R_1 should also be an acquire (since data $\rightarrow$ acquire ordering is not essential). Proof is from Lemma 3 (treating R_1 as the acquire, first form applies).

Case 2: $W \rightarrow R$, where W is a write. Proof is from Lemma 3, second form applies.

Case 3: $R \rightarrow A$, where A is any access. Proof is from Lemma 3, first form applies. ■

4. Implementation

In this section we present two algorithms for identifying synchronization reads, as used in our implementation. The first algorithm (**Fast**) only identifies acquires that meet our control signature, while the second (**Safe**) is conservative, as it additionally identifies acquires that only match our address signature.

While conservatism demands application of the address signature, in practice we find that only the control signature is required. In all the experiments we perform (see Section 5) we find no acquires that only meet the address signature. To reinforce this point we performed an empirical study of 9 common synchronization primitives, the results of which are presented as Table 2. It is worth noting that these primitives represent common patterns used in synchronization, indeed some underpin programs we examine later in Section 5. As we can see, acquires that match the control signature are far more prevalent. While there are acquires that meet the address signature, all of those also meet the control signature.

	Acquires			
	Addr	Ctrl	Pure Addr	Source
Chase Lev WSQ	✓	✓	✘	[12]
Cilk-5 WSQ	✘	✓	✘	[18]
CLH Lock	✓	✓	✘	[14]
Dekker	✘	✓	✘	[16]
Lamport	✘	✓	✘	[27]
MCS Lock	✓	✓	✘	[32]
Michael Scott LFQ	✓	✓	✘	[33]
Peterson	✘	✓	✘	[35]
Szymanski	✘	✓	✘	[39]

Table 2. Breakdown of the types of acquires found in common synchronization kernels. Notably, no acquires are found to only meet the address signature.

We make one simplifying assumption in our implementations, this is that the synchronizing reads occur in the same function as the condition to which they lead. While an interprocedural algorithm would be a necessary step to achieving soundness, such a guarantee would also require access to all libraries/functions used, at compile time. We believe that this assumption is reasonable, since it is extremely rare for these two operations, which intuitively form the synchronization, to be split across two functions (although it is possible to construct a contrived example). Indeed in none of the implementations of the primitives examined (implementations for CLH Lock and MCS Lock from [15], all others from [5]), nor the real programs examined in Section 5 is this separation found.

Both of the algorithms depend on an intraprocedural static slicer that performs the actual identification of the synchronizing reads, this is presented in Section 4.1. All the algorithms operate on infinite register load-store intermediate representations. We will now examine each algorithm in detail, before finally outlining the generation of orderings and the fence minimization algorithm to which we input them. We assume that the set of escaping loads and stores has previously been identified, using a thread-escape analysis as in Pensieve.

4.1 Identifying Control Acquires

The algorithm for identifying escaping reads that match our control signature (**Fast**) is presented as Listing 1. To determine reads that meet our control signature we must determine which reads have branches (conditions) in their forward slice. To determine this efficiently, the algorithm in fact focuses on each conditional branch and examines the reads in its backwards slice. For each conditional branch in a function we retrieve the instructions that define the branch operands (lines 8 and 9). Then we initiate the backwards slicer to populate *sync_reads* with escaping loads from the backwards slice of the conditional branch, line 11.

```
1  sync_reads = ∅
2  seen = ∅
3
4  for cond_branch in function
5  {
6     work_list = ∅
7
8     for operand in cond_branch
9        work_list.insert(get_def(operand));
10
11    slicer(&work_list, &seen, &sync_reads);
12 }
```

Listing 1. Algorithm Fast, for matching the control signature.

Backwards Slicing - The algorithm for backwards slicing and populating *sync_reads* is presented as Listing 2. This algorithm performs a conservative intraprocedural backwards slice from the initial contents of *work_list*. Every load found while processing the *work_list* is compared against the results of the prior escape analysis (line 14), and if escaping, added to *sync_reads* (line 15).

To ensure conservatism, whenever a load is found, alias analysis is used to find all stores in the function that potentially wrote the value being read (line 17). These stores are added to the *work_list* to be processed later. For instructions that are not a load, each operand is processed and the defining instructions of those operands are added to the *work_list* (lines 22 and 23).

To avoid becoming trapped in cycles and to improve efficiency, both of the signature matching algorithms maintain sets of previously examined instructions, *seen*. The slicing algorithm is responsible for populating (line 10) and checking against (line 7) these sets. Once the *work_list* has been exhausted, the algorithm terminates.

```
1  slicer (*work_list, *seen, *sync_reads)
2  {
3     while (!work_list->empty())
4     {
5        inst = work_list->first();
6        work_list->remove(inst);
7        if (seen->count(inst))
8           continue;
9
10       seen->insert(inst);
11
12       if (inst.is_load())
13       {
14          if (escaping_reads.count(inst))
15             sync_reads->insert(inst);
16
17          for store in potential_writers(inst)
18              work_list->insert(store);
19       }
20       else
21       {
22          for operand in inst
23             work_list->insert(get_def(operand));
24       }
25    }
26 }
```

Listing 2. Algorithm for backwards slicing and the registration of escaping reads contained in the slice.

4.2 Identifying Both Control and Address Acquires

As we previously stated, the algorithm presented in the previous sections provides sufficient coverage for all the real programs we have seen. It is however possible that an acquire only meets the address signature. To contend with this eventuality we develop a conservative variant of our algorithm (**Safe**), presented as Listing 3. This variant identifies escaping reads that meet either or both of the signatures identified.

As with the algorithm for the control signature, we use a backwards slice. In addition to conditional branches, the slicing is performed from every instruction that is either a dereference or an address calculation. This ensures that any escaping reads that contribute to a value used as an address are added to *sync_reads*. In the case of a dereference, the slicer is applied to the operand of the instruction, i.e., the address (line 16). In the case of an address calculation (for example a *GetElementPtr* instruction in LLVM IR), the offset is sliced (line 13). As is to be expected, these two cases often overlap with an address calculation in the backwards slice and therefore subordinate to a dereference. Here again, the use of the *seen* set prevents reiteration.

```
1  sync_reads = ∅
2  seen = ∅
3
4  for inst in function
5  {
6     if (inst.is_address_calculation() or
7        inst.is_dereference() or
8        inst.is_cond_branch())
9     {
10       work_list = ∅
11
12       if (inst.is_address_calculation())
13          work_list.insert(get_def(
14             inst.offset()));
15       else
16          work_list.insert(get_def(
17             inst.operand()));
18
19       slicer(&work_list, &seen, &sync_reads);
20    }
21 }
```

Listing 3. Algorithm Safe, that identifies escaping reads that match either signature.

4.3 Generating Pruned Orderings

Whichever algorithm has been used to populate *sync_reads*, the next step is the generation of orderings. Ordering generation is done in line with Pensieve, generating an ordering for every pair of variables in the set of potentially escaping loads and stores, if there exists a path between them. Within a basic block the order of statements gives a directed linear sequence of accesses. Whether there exists a path between basic blocks is determined prior to this process with an examination of the CFG, to create a lookup table of reachability. This can then be queried during ordering generation.

The addition that we make to ordering generation is to prune $w \rightarrow r$ and $r \rightarrow r$ orderings which do conform to $w \rightarrow r_{acq}$ and $r_{acq} \rightarrow r$ respectively. The pruning is achieved by querying orderings of the form $w \rightarrow r$ and $r \rightarrow r$ for previously identified synchronizing reads.

4.4 Fence Minimization

Given the set of orderings to enforce, a fence minimization algorithm is used to place as few fences as possible, while still enforcing all required orderings. To place fences, we use the locally-optimized fence placement algorithm described in Fang et al. [17]. The only alteration we make to this algorithm is to not automatically place a fence at the beginning

of each function, such a fence is only placed if the function contains synchronizing reads. The rationale for placing this fence is to enforce interprocedural orderings, under x86-TSO if the function contains no synchronizing reads then no interprocedural $w \rightarrow r$ orderings can terminate within the function and the absence of a full fence does not affect correctness.

When determining full fence placement we need only consider orderings that the hardware will not enforce. Our technique is generally applicable, but in our experiments we target x86-TSO and therefore we only consider orderings of the form $w \rightarrow r$, as the other orderings are enforced automatically by hardware. However, to prevent incorrect reorderings by the compiler, we place compiler directives to enforce orderings of any other form. Specifically, these directives take the form of empty memory-clobbering assembly instructions which have no presence in the final binary but prevent reordering of memory related statements around them. The same minimization algorithm is used here, with the decision as to whether to place a full fence or a compiler directive determined by whether the set of orderings that would be enforced contains one of the form $w \rightarrow r$.

5. Results

We implemented our algorithms and a locally-optimized fence minimization algorithm based on Fang et al. [17], in LLVM 3.4.1. The programs were all compiled using the $O2$ optimizations.

Using a set of lock-free programs and the SPLASH-2 [43] benchmarks, we compare both the **Fast** (control acquires only) and **Safe** (control and address acquires) variants of our approach with an implementation of Pensieve[5] using locally-optimized fence minimization (as described in Fang et al. [17]). To establish a performance baseline we also compare against a (minimal) manual fence placement. The lock-free programs are introduced in Table 3.

Canneal	A kernel that seeks to minimize routing cost for chip design using cache-aware simulated annealing. This program was drawn from the PARSEC suite [8], and was run with the Simlarge input set.
Matrix	A parallel implementation of matrix multiplication, that takes in two matrices and outputs both potential matrix products. To allow 64 threads to compete for work, it is built on top of a lock-free queue as described by Michael & Scott [33]. It was applied to two square matrices both of dimension 1,024.
SpanningTree	An implementation of a parallel spanning tree algorithm, built on top of a work-stealing queue as described by Bader et al. [6]. It was applied to a graph of 10,000 nodes, each of degree 1,000.

Table 3. Descriptions of the lock-free programs used.

It is worth noting that the programs considered are well-synchronized because they employ user-defined synchronization[6] and hence require fences on relaxed models for correctness.

Our results are organized as follows. Firstly, we examine how many reads marked as potentially thread-escaping that our algorithms mark as an acquire, giving us a measure of the effectiveness of our technique. Secondly, we compare and breakdown by type the number of orderings generated by the naive and both variants of our approach. Thirdly, we present the reductions in the number of full memory fences placed for an x86-TSO machine, where only orderings of the form $w \rightarrow r$ require such enforcement. Finally, we present the performance improvements achieved over Pensieve. For the performance experiments, we used an Intel i3-2100 running Linux 3.2.0-67 (Ubuntu 12.04.4). All the programs were run using 64 threads.

5.1 Synchronization Read Detection

Applying the algorithms as defined in Section 4, we are able to mark a subset of the potentially escaping reads as acquires. The percentage of these reads that are marked acquires by each variant of our approach is presented as Figure 7.

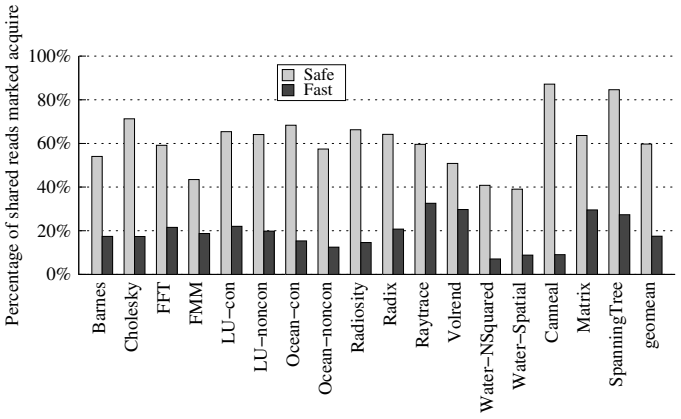

Figure 7. Static percentage of potentially thread-escaping reads that our analysis marks as an acquire.

As we can see, the Fast form of our analysis is able to greatly reduce the number of reads which must be treated as acquires. In the best case (*Water-NSquared*), only 7% are potentially acquires. On average[7] we see 18% of the reads marked as acquires. Even in the worst case our analysis is able to significantly reduce the number of reads that must be treated as acquires. We see this in *Raytrace*, with 33% marked as acquires.

[5] We use the term Pensieve throughout this section to refer to the version presented in Fang et al. [17] with locally-optimized fence minimization, rather than the later Sura et al. [38].

[6] While the lock-free programs use user-defined synchronization exclusively, the SPLASH-2 programs make use of both user-defined synchronization (in programs such as FMM [40] and Volrend [34]), and also employ library calls to locks and barriers.

[7] Geometric mean is used for all normalized results.

Using the Safe variant, we are still able to reduce the number of reads marked as acquires in all cases. On average we see 60% marked as acquires. In the best case (*Water-Spatial*), only 39% need be marked.

5.2 Ordering Pruning

Using the acquire detection results, we are able to prune the orderings considered by the fence placement algorithm. As detailed in Section 2.3, identifying acquires allows pruning of those $w \rightarrow r$ and $r \rightarrow r$ orderings that do not conform to the rules in Table 1. Figure 8 presents the results of this pruning.

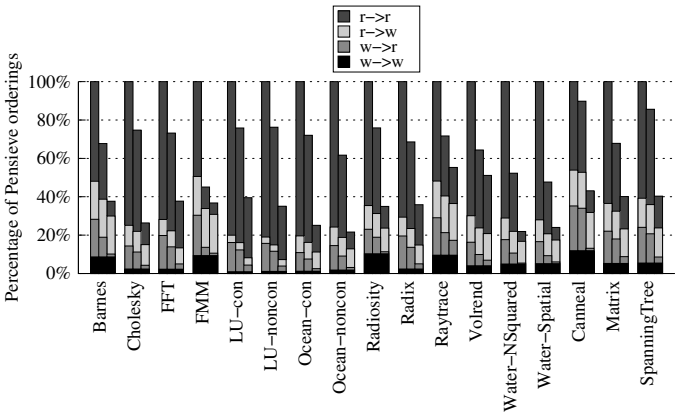

Figure 8. A breakdown of orderings by type for Pensieve (left), Safe (center), and Fast (right).

As Figure 8 shows, our Fast approach significantly reduces the number of $w \rightarrow r$ and $r \rightarrow r$ orderings required to be considered for fence placement. This result holds across all the programs tested, with an average of 34% orderings remaining after application of our approach. As $r \rightarrow r$ orderings form the majority of orderings in all but two of the programs, reducing them has the largest overall impact on the number of orderings considered. $w \rightarrow r$ orderings are also pruned significantly, though as they often form only a small percentage of overall orderings, the impact of this on the total number of orderings is smaller. As we do not identify a specific subset of writes as releases, $r \rightarrow w$ and $w \rightarrow w$ orderings are unaffected by the pruning process. With $w \rightarrow r$ and $r \rightarrow r$ orderings forming the majority of the orderings, the correlation between the percentage of reads marked as acquires (Figure 7) and the percentage of orderings that survive pruning is not unexpected.

Examining the results for the Safe variant, we see that reductions in $w \rightarrow r$ and $r \rightarrow r$ are still achieved. Specifically, only 68% orderings remain on average.

5.3 Fence Placement

In placing fences, we consider the requirements of an x86-TSO hardware model. Here, only $w \rightarrow r$ orderings require enforcement by a full memory fence. Other orderings are automatically enforced by the hardware and are enforced during the compilation process with empty memory-clobbering assembly instructions, that have no presence in the final program. As Figure 8 showed, our pruning was very effective at reducing the number of $w \rightarrow r$ orderings.

Applying the fence minimization algorithm to the pruned sets of orderings for both variants of our approach and Pensieve for comparison, we determine the percentage of full fences that are still placed when using pruned orderings. This is shown as Figure 9.

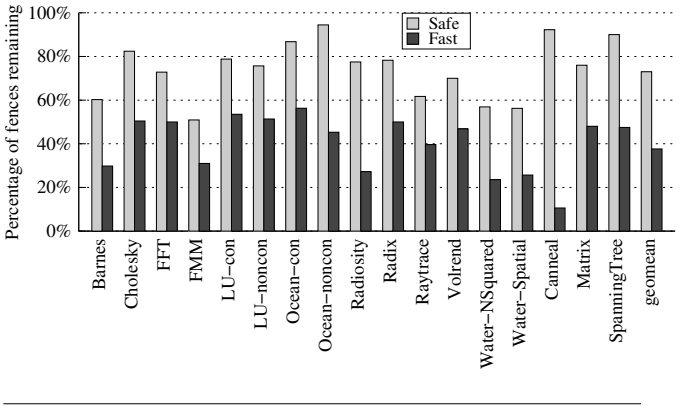

Figure 9. Static percentage of full fences that remain on x86-TSO after using pruned orderings.

As Figure 9 shows, the impact of pruning orderings is significant in reducing the static number of fences that the algorithm places to enforce $w \rightarrow r$ orderings. As we can see, the percentage of fences placed is quite strongly correlated with the percentage of reads marked as acquires (Figure 7). For the Fast algorithm we see on average 38% of Pensieve's fences required, with *Canneal* receiving a 89% reduction in the number of fences placed. For the Safe variant, on average 73% of Pensieve's fences are required.

5.4 Performance Improvements

To examine the impact of reducing the number of fences, we executed the programs having applied Pensieve, both variants of our approach and normalize these against manual fence placement. Each of the experiments was repeated 100 times and averages taken. The results of these experiments are presented as Figure 10.

As we can see, in all cases the fences placed using either variant of our approach results in a performance improvement over using a naive set of orderings. On average we see that Pensieve is 1.94x slower than the baseline, with our Fast approach being only 1.44x slower than the baseline. The Safe approach is 1.69x slower than the baseline. In other words, on average, our Fast approach results in a 30% speedup over Pensieve, while the Safe approach results in executions 14% faster than Pensieve. In the best case (*Matrix*) we achieve a 90% improvement over Pensieve using

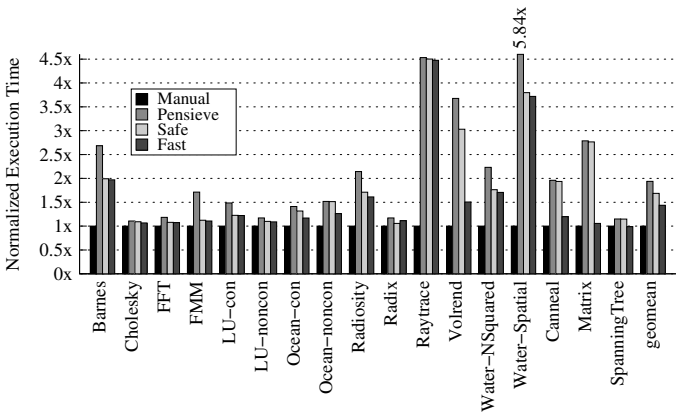

Figure 10. Execution time with fences placed using Pensieve, Safe, and Fast, normalized against manual fence placement.

Fast. For the Safe approach, the best case (*Water-Spatial*) is 42% faster than Pensieve.

Examining the performance results for individual programs, we see that the speedups achieved over the naive are not strongly correlated with the changes in static fence placement. This is due to specific fences being reached more than others during the execution of the program. This is best highlighted by the case of *Raytrace*, where significant reductions in the number of static fences is not reflected in performance improvement. When looking at the results for Safe, we see that in some cases it is closer to Pensieve (e.g., *Ocean-noncon*) and in others (e.g., *Water-Spatial*) closer to Fast. To which result Safe is most similar depends on the propensity of the use of escaping reads as addresses in heavily executed code regions. In one program (*Radix*), we see Safe outperforming the simple algorithm. This is likely due to the short running time and small number of fences placed, making the result susceptible to noise. This also accounts for why Fast achieves a 1% improvement over the baseline for *SpanningTree*.

In terms of performance comparison with the manual baseline, we see that there is still some improvement possible. There are two reasons for this discrepancy. First is the difficult orthogonal problem of optimal fence minimisation given a set of orderings to enforce. In extremis this may even require profiling to determine the fence insertion points that have the minimal impact on performance. Secondly, while our signatures significantly prune the number of shared reads considered as acquires, some false positives still remain.

6. Related Work

Programmer-centric memory models Adve and Hill [3] and Gharachorloo [22] were the first to propose programmer centric memory consistency models, where the system enforces SC as long as the programmer writes data-race-free (DRF) programs and provides information about synchronization operations. Indeed Adve's DRF based models [1] and Gharachorloo's PL based models [20] are the precursors to the memory consistency models adopted by languages such as C [10] and Java [30]. The main difference between the above works and ours is that, while they assume programmer-annotated synchronization labels, we assume unlabeled data-race-free programs.

Delay-set analysis Shasha and Snir [36] were the first to consider the problem of computing the minimum number of memory orderings (delays) to ensure that a concurrent shared memory program satisfies SC. In this work, we focus on how the above orderings can be pruned if the shared memory program is a DRF (but unlabelled) program. To put it succinctly, we do Delay-set analysis for unlabelled DRF programs.

A more recent work [5] attempts to address the scalability issues inherent in Delay-set analysis by examining an over-approximation of the critical cycles. It is however limited in failing to handle recursion and dynamic thread creation, the latter of which is common in the programs examined in our evaluation. Specifically, this tool does not handle *pthread_create* calls in loops that could not be statically unrolled. We note, however, that our signatures would be equally applicable to [5] and our choice to build on top of Pensieve is due to its lack of the limitations described above.

Fence minimization There have been a number of works [17, 25, 42] which focus on computing the minimal number of fences for satisfying the orderings given by Delay-set analysis. These works are orthogonal to our work, as these can very well be applied for satisfying the pruned orderings given by our analysis.

Synchronization detection Our work is related to prior work [40, 41, 44] on busy-wait synchronization detection. Tian et al. [40, 41] proposed a dynamic analysis technique for identifying user-defined busy-wait synchronizations. Since the above work uses dynamic analysis, they suffer from false negatives – in other words, some synchronizations can be missed. Subsequently, Xiong et al. [44] showed how synchronizations can be identified using static analysis, so that there can be no false negatives. Our work differs from the above in one important aspect. The above analysis is only applicable for busy-wait synchronization; thus it will miss identifying acquires used in non-blocking algorithms such as those used in our evaluation. It is worth noting that missing such acquires leads to correctness issues in our context which explains why the above detectors cannot be used in the context of our work. Indeed, one of the nice side-effects of our work is that to the best of our knowledge, ours is the first general acquire detector.

Hardware based memory ordering There have been a number of recent works [9, 21, 23, 29, 37] which have proposed techniques for efficiently enforcing memory ordering. In contrast with the above works each of which involve hardware support, we do not use any hardware support. Further-

more, each of the above works are orthogonal to us, in that, they can very well be used to efficiently enforce the pruned orderings given by our work.

SC-preserving compiler Ahn et al. [4] proposed the Bulk compiler which together with Bulk hardware (which enforces hardware SC at chunk level) guarantees SC at the language level. In other words, the Bulk compiler preserves SC by ensuring that it does not reorder memory operations across chunks. More recently, Marino et al. [31] proposed the SC-preserving compiler which together with SC hardware (which enforces SC at the hardware level) guarantees SC at the language level. Their main result is that it is possible for the compiler to preserve SC without significant slowdown (<5% on average across a suite of parallel programs). On the other hand, they assume that the hardware cannot reorder operations, i.e., they assume that the hardware enforces SC. In contrast, our work considers the problem of how to enforce SC on hardware that could reorder memory operations. Of course, to preserve SC at the language level we would need a compiler that preserves SC (i.e., the above works). Recall that in our implementation we ensure that the compiler cannot reorder shared memory operations by inserting an empty memory-clobbering assembly instruction between such operations, which LLVM interprets as a compiler fence. It is worth noting that this corresponds to the naive-SC variant [31]. We could have very well used the SC-preserving compiler proposed (with all optimizations), which could potentially translate into better performance. In this respect, our work is orthogonal to the above works.

7. Conclusions

Relaxed hardware memory consistency models are used to ensure performance in multicore computers. A large body of legacy code assumes SC. Placing sufficient but minimal fences is challenging. The starting point of understanding the required placement is Delay-set analysis. However, in practice approximations are applied, resulting in many superfluous orderings.

With Delay-set analysis too hard in the general case and with languages converging to DRF based memory models, we for the first time attack the problem of Delay-set analysis for legacy DRF programs. We prove that a read of shared data must match at least one of two signatures to be an acquire. We determine that this enables the pruning of a large number of orderings, reducing the set that need be considered for fence placement.

Developing both simple (control acquires) and conservative (control and address acquires) algorithms, we implement them in LLVM and demonstrate the significance of our contribution. Applying our control acquire detection on a set of lock-free programs and to SPLASH-2, we reduce the average number of orderings considered by 66%. Using a fence minimization technique, this translates to an average of 62% fewer fences on x86-TSO and up to 2.64x speedup over an existing practical technique.

References

[1] Sarita Vikram Adve. 1993. *Designing memory consistency models for shared-memory multiprocessors*. Ph.D. Dissertation. University of Wisconsin, Madison, WI, USA.

[2] Sarita V. Adve and Kourosh Gharachorloo. 1995. Shared Memory Consistency Models: A Tutorial. *IEEE Computer* 29 (1995), 66–76.

[3] Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering - A New Definition. In *ISCA*. 2–14.

[4] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S.Midkiff, and David Wong. 2009. BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support. *IEEE Micro* (2009).

[5] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2014. Don't Sit on the Fence - A Static Analysis Approach to Automatic Fence Insertion. In *CAV*. 508–524.

[6] David A. Bader and Guojing Cong. 2005. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *J. Parallel Distrib. Comput.* 65, 9 (2005), 994–1006.

[7] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL*. 55–66.

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *PACT*.

[9] Colin Blundell, Milo M. K. Martin, and Thomas F. Wenisch. 2009. InvisiFence: performance-transparent memory ordering in conventional multiprocessors. In *ISCA*.

[10] Hans-Juergen Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *PLDI*.

[11] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. 2007. BulkSC: bulk enforcement of sequential consistency. *SIGARCH Comput. Archit. News* 35, 2 (2007), 278–289.

[12] David Chase and Yossi Lev. 2005. Dynamic Circular Work-stealing Deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '05)*. ACM, New York, NY, USA, 21–28.

[13] Daniel Chazan and Willard Miranker. 1969. Chaotic relaxation. *Linear algebra and its applications* 2, 2 (1969), 199–222.

[14] Travis Craig. 1994. *Building FIFO and priority-queuing spin locks from atomic swap*. Technical Report. Technical Report 93-02-02, University of Washington, Seattle, Washington.

[15] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*. 33–48.

[16] E. W. Dijkstra. 1965. Solution of a Problem in Concurrent Programming Control. *Commun. ACM* 8, 9 (Sept. 1965), 569–.

[17] Xing Fang, Jaejin Lee, and Samuel P. Midkiff. 2003. Automatic fence insertion for shared memory multiprocessing. In *ICS*. 285–294.

[18] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. *SIGPLAN Not.* 33, 5 (May 1998), 212–223.

[19] Andreas Frommer and Daniel B Szyld. 2000. On asynchronous iterations. *Journal of computational and applied mathematics* 123, 1 (2000), 201–216.

[20] Kourosh Gharachorloo. 1995. *Memory Consistency Models for Shared-Memory Multiprocessors*. Ph.D. Dissertation. Stanford University.

[21] Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy. 1991. Two Techniques to Enhance the Performance of Memory Consistency Models. In *ICPP (1)*. 355–364.

[22] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1990. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *ISCA*. 15–26.

[23] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. 1999. Is SC + ILP=RC?. In *ISCA*. 162–171.

[24] Mark D. Hill. 1998. Multiprocessors Should Support Simple Memory-Consistency Models. *Computer* 31, 8 (1998), 28–34.

[25] Amir Kamil, Jimmy Su, and Katherine Yelick. 2005. Making Sequential Consistency Practical in Titanium. In *SC*. IEEE, Washington, DC, USA, 15.

[26] L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programm. *IEEE Trans. Comput.* 28, 9 (1979), 690–691.

[27] Leslie Lamport. 1987. A Fast Mutual Exclusion Algorithm. *ACM Trans. Comput. Syst.* 5, 1 (Jan. 1987), 1–11.

[28] Jaejin Lee and David A. Padua. 2001. Hiding Relaxed Memory Consistency with a Compiler. *IEEE Trans. Comput.* 50, 8 (2001), 824–833.

[29] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. 2010. Efficient sequential consistency using conditional fences. In *PACT*. ACM, 295–306.

[30] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *POPL*. ACM, New York, NY, USA, 378–391.

[31] Daniel Marino, Abhayendra Singh, Todd D. Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A Case for an SC-Preserving Compiler. In *PLDI*.

[32] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (Feb. 1991), 21–65.

[33] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*.

[34] Adrian Nistor, Darko Marinov, and Josep Torrellas. 2010. InstantCheck: Checking the Determinism of Parallel Programs Using On-the-Fly Incremental Hashing. In *MICRO*. 251–262.

[35] Gary L. Peterson. 1981. Myths About the Mutual Exclusion Problem. *Inf. Process. Lett.* 12, 3 (1981), 115–116.

[36] Dennis Shasha and Marc Snir. 1988. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (1988), 282–312.

[37] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd D. Millstein, and Madanlal Musuvathi. 2012. End-to-end sequential consistency. In *ISCA*.

[38] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. 2005. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *PPoPP*. ACM, New York, NY, USA, 2–13.

[39] B. K. Szymanski. 1988. A Simple Solution to Lamport's Concurrent Programming Problem with Linear Wait *(ICS '88)*. ACM, New York, NY, USA, 621–626.

[40] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. 2008. Dynamic recognition of synchronization operations for improved data race detection. In *ISSTA*. 143–154.

[41] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. 2009. Automated dynamic detection of busy-wait synchronizations. *Softw., Pract. Exper.* 39, 11 (2009), 947–972.

[42] Chi-Leung Wong, Zehra Sura, David A. Padua, Xing Fang, Jaejin Lee, and Samuel P. Midkiff. 2002. The Pensieve Project: A Compiler Infrastructure for Memory Models. In *ISPAN*. 239–244.

[43] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*. 24–36.

[44] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. 2010. Ad Hoc Synchronization Considered Harmful. In *OSDI*. 163–176.